CS 3110 Problem Set 6: *AGE OF UPSON*

Assigned: 12 April 2012

Final submission due: 3 May 2012 **(no extensions)**
Design meetings: April 18 - 19

CS3110 Spring 2012

1 Updates

1. Updated the description of One Game Step to clarify building creation, and that Villagers should use their `COOLDOWN` constant for building, whereas other units should use it for attacks.
2. Updated Game Rules to include clarifications on the dimensions of some of the constants.
3. Added clarifications on how score is kept track of by using `cKILL_UNIT_SCORE`, `cKILL_BUILDING_SCORE`, and `cRESOURCE_COLLECTED` or `cADVANCED_RESOURCE_COLLECTED`.

2 Introduction

In this assignment, you will develop a game called *Age of Upson*. There are few constraints on how you must implement this project. This freedom does not mean you can abandon what you have learned about abstraction, style and modularity; rather, this project is an opportunity to demonstrate all three in the creation of elegant code.

You should start by carefully designing your system and presenting it at a *design meeting* where you will meet with a course staff member to discuss your design. Part of your score will be based on the design you present at this meeting.

On 12 May, after the final exam, there will be an *Age of Upson* tournament. We highly encourage you to submit your bot programs for this tournament. There will be free food and the chance to watch your bot perform live. The winner gets bragging rights and will have their name posted on the 312/3110 Tournament hall of fame. Rumor has it there will also be potentially awesome prizes involved!

2.1 Reading this document

The types referenced in this document that are not primitive to OCaml are defined in `definitions.ml`. Constants follow the following naming convention: They begin with a lowercase `c`, and the rest is a descriptive name of the constant in all caps. For example, `cBOARD_WIDTH` specifies the pixel width of the playing board. The reason for this mysterious `c` in front of all the constants is that OCaml actually doesn't allow value names to begin an uppercase letter—only type constructors can do that.

Note that some constants may not make sense initially. This serves two purposes. It ensures that your implementation of the game does not depend on any specific descriptions of the constants but only the constant itself. Second, it allows us to modify the constants to create a balanced tournament environment.

2.2 Updates to Problem Set

Any updates to the problem set will be marked in red.

2.3 Point Breakdown

- Design meeting – 5 pts
- Game – 50 pts
- Bot – 40 pts
- Documentation and design – 5 pts

3 Game Rules

3.1 General Rules

The game begins with each team starting with `cSTARTING_VILLAGER_COUNT` Villagers on their team, and one Town Center. From here, the player must use their Villagers to collect resources, and build a Barracks so that they can start generating military units. The player can then choose to upgrade to a more advanced age to create stronger military units. The goal is to use these military units to kill all the opponent's units or to destroy the opponent's Town Center. The first team to do either of these wins! There is also a time limit specified by `cTIME_LIMIT` in seconds, if no team has won by the end of this time, the team with the highest score wins.

3.2 The Units

There are 7 different type of units. Each of the different `unit_types` has several constants:

- `ATTACK`: the damage the unit deals to other units and buildings.
- `HEALTH`: the unit's starting health
- `SPEED`: the unit's moving speed. The speed of a unit is per pixel, not per tile.
- `RANGE`: the range at which the unit can attack other units or buildings. Look into the util functions we provided to convert the range constants into a pixel distance.
- `COOLDOWN`: the time the unit must wait before it can attack again. For Villagers, cooldown is the time it takes the villager to construct a building. All cooldowns are in seconds.
- `COST`: the amount of resources required to create a unit of this type, specified as a tuple of (food, wood) costs.

Archer and Elite Archer

The Archer is a ranged unit, capable of attacking other units and buildings from afar with arrows. The Elite Archer is the upgraded version of the Archer. Both the Elite Archer and Archer obtain an attack bonus against Pikemen and Elite Pikemen.

Knight and Elite Knight

The Knight is a powerful, fast unit, which you can use to make quick attacks. The Elite Knight is the upgraded version of the Knight. Both the Elite Knight and Knight obtain an attack bonus against Archers and Elite Archers.

Pikeman and Elite Pikeman

The Pikeman is the standard unit which you can use to begin building up your army. The Elite Pikeman is the upgraded version of a Pikeman. Both the Elite Pikeman and Pikeman obtain an attack bonus against Knights and Elite Knights. The Pikeman is the *only* military unit that can be created in the Dark Age (ages will be explained in section 3.5). All other units, including the Elite Pikeman, can only be created in the Imperial Age.

Villager

The Villager is the only unit that can be used for collecting resources and constructing new buildings. However, a Villager cannot attack. As such, it does not have an ATTACK or RANGE constant. Also, as discussed above, the Villager's cooldown refers to the time it takes to construct a building, as opposed to all other units for which cooldown refers to the time before it can attack again. The Villager, along with the Pikeman, can be created in the Dark Age.

3.3 The Buildings

There are two types of buildings. Each Building has specific constants which are relevant to it.

- `HEALTH`: the building's starting health.
- `COST`: the amount of resources required to create this building, specified as a tuple of (food, wood) costs.

Town Center

The Town Center is a player's starting building, and no further Town Center's can be created. Given that a player cannot create additional Town Centers, there is no constant that represents the Town Center's cost. The Town Center is a player's most important building and is from where Villagers are created. A player loses if the player's Town Center is destroyed.

Barracks

The Barracks is a player's military center, from which all military units (i.e. all units that are not Villagers) are created. Multiple Barracks can be created.

3.4 The Upgrades

There are 4 upgrades that the player can do throughout the game. During the Dark Age, there is only one possible upgrade: Age Upgrade, to change from the Dark Age to the Imperial Age. Once in the Imperial Age, the player has access to three more upgrades: the Unit Upgrades. There is one Unit Upgrade per military unit to upgrade a unit to its Elite version. When a unit has upgraded to its Elite version the basic version of the unit can no longer be created. Each Upgrade has a constant specifying its `COST` as a tuple of (food, wood) costs.

3.5 The Ages

There are two ages that the player can go through throughout the game. Each player starts in the Dark Age, where Barracks can be built, and only Pikemen and Villagers can be created. The only Upgrade available is the Age Upgrade to move onto the Imperial Age. Once in the Imperial Age, the player can do all other upgrades, and create all other units.

3.6 The Resources

On the board you will see berry bushes and trees, each of which are used to collect food and wood, respectively. These are the resources you need to collect to create units and buildings, and to do upgrades. The initial locations of these resources are specified in the constants as `FOOD_TILES` and `WOOD_TILES`, which are lists of (row, column) tuples. Villagers move to these resources in order to collect resources and increase their team's food and wood count.

3.7 The Board

The board pixel size is defined by a BOARD_WIDTH and a BOARD_HEIGHT. Furthermore, the board can be divided into tiles, with NUM_X_TILES columns and NUM_Y_TILES rows. The size of each tile in pixels is specified by TILE_WIDTH and TILE_HEIGHT. The locations of resources and buildings are specified as tiles, described by (row, column) tuples. The locations of units are specified as (x, y) coordinates, representing an actual pixel on the board. Each building takes up 4 tiles (2 columns and 2 rows), whereas each resource takes up only 1 tile. Buildings cannot be placed on top of resources.

4 Game Module

This module is in charge of implementing all of the rules of the game. The game server calls the functions of this module in order to progress the game.

- `initGame`: This initializes the game into a game type. This contains all of the information for a game which has yet to begin.
- `initUnitsAndBuildings`: This initializes the units and buildings of your game type. If we divide the board in half, then the red team starts with a Town Center at an arbitrary location on the left half, and the blue team starts with a Town Center at an arbitrary location on the right half. In addition, each team starts with cSTARTING_VILLAGER_COUNT Villagers arbitrarily placed around their Town Center.
- `startGame`: This tells the game that all initialization is done and the game is beginning.
- `handleAction`: This is the method by which the AI will send commands. This method is in charge of updating the state of your game to reflect AI commands.
- `handleTime`: The server will call this function in order to update a game by a time step.
- `handleStatus`: This is the method by which the AI can request the status of the game, team, units, buildings, and resources. This method is in charge of returning the requested information.

  Note: The game object which is passed to all of these functions is the one that was originally generated by initGame. Therefore, all modifications that you make to the game can not be done functionally. This means you need to be very careful when modifying the game object!

One Game Step

In `handleTime`, you should follow this guide in order to ensure that *Age of Upson* is played correctly.

1. First, check to see if the game is over. The game is over when the game has gone on for cTIME_LIMIT, or one of the teams has had all of their units killed, or one of the teams has had their Town Center destroyed. In the case of a time out, return the team with the higher score. If the game does not end, progress one step.
2. Handle all attacks in the game. A player can queue multiple attack targets for each unit, so we recommend that you keep an attack queue for each unit. At each time step, carry out the first attack in the queue for each unit. If it is an invalid attack (if the target is out this unit's range, or if the target is already dead, and thus no longer on the board), then discard it and continue down the queue until you find the first legal attack which can be carried out. If there is no legal attack in the queue then this unit does no attack, and its attack queue should be empty at this point. Update any necessary timers to handle the attack cooldowns.
3. Remove all dead units and buildings from the game. Increase each team's score by cKILL_UNIT_SCORE and cKILL_BUILDING_SCORE for every unit and building they killed.
4. Remove all resources (berry bushes and trees) whose count has reached 0 (meaning that Villagers have collected all available resources for this berry bush or tree).

5. Handle building creation. Update the timers of all Villagers that are currently building, and add the building they were working on to the GUI once the time is complete. Villagers use `cVILLAGER_COOLDOWN` as their building timer. Villagers should not move while they are building.

6. Update the position of all of the units. We recommend that you keep a queue of positions for each unit, and at each step just move each unit to the next position in its queue. If a unit does not have a position to move to, then keep the unit still.

5 Communication

5.1 Client-Server Framework

Age Of Upson makes use of a client-server framework. Under this framework, the game server is responsible for keeping track of the game state, applying the game rules, and so on. Clients (i.e., players) are run as an entirely separate processes and can keep track of whatever information they want, but they need to send messages to the server to perform game actions or receive information. Players communicate with the game server by sending information over channels. The protocol for messages is defined by the type `action` in `definitions.ml`.

There are nine types of action messages defined for the client to communicate to the server. Each action should be handled in `handleAction` in `game.ml`, and should return either a `Success` or `Failed` result. These actions are:

- `QueueAttack(unit_id, target)`: The client uses this to tell a unit that it should add the given target (either a building or another unit) to its attack queue. If the target is already dead, or is of the same color as the unit doing the attack, then you should return `Failed`, otherwise queue the attack. This should also fail if the unit queuing the attack is a Villager.

- `QueueCollect(unit_id)`: The client uses this to tell a given unit to collect resources. If the unit is not a Villager, or is not standing on a resource to be collected, then you should return `Failed`, otherwise handle the resource collection. This should also fail if the unit performing the collection is not a Villager. When you handle the resource collection, the amount of resources collected are also used to add to a team's score, so make sure to either add `cRESOURCE_COLLECTED` or `cADVANCED_RESOURCE_COLLECTED` to the team's score, depending on which age the team is in.

- `QueueMove(unit_id, target_pos)`: This tells the unit to queue a new target location. Keep in mind that the position being sent is a *target* location, it is not the position that the unit will be in at the next time step. This means that all intermediate positions between the unit's current position and the target position should be queued. We recommend that you take into account the unit's speed to use as an interval when you generate the list of positions between the unit's current position and target position. Then add all of these intermediate positions to the unit's move queue.

- `QueueBuild(unit_id, building)`: This tells the game to add a building at the unit's current location. This should fail if the building will overlap with any type of resource (keep in mind that buildings occupy 2x2 tiles), or if the player does not have enough food or wood to add this building. This should also fail if the unit performing the build is not a Villager, or if the building to be added is a Town Center (players should not be allowed to build Town Centers).

- `QueueSpawn(building_id, unit)`: This tells the game to add a unit at the current building. This should fail if the player does not have enough food or wood to create this unit. This should also fail if the player is trying to spawn a unit without having done the necessary Upgrades, or if the player is trying to spawn a unit at the wrong building (e.g. trying to create a Villager at a Barracks)

- `ClearAttack(unit_id)`: This clears a given unit's attack queue.

- `ClearMove(unit_id)`: This clears a given unit's move queue.

- `Upgrade(upgrade_type)`: This carries out the specified upgrade, such as upgrading to the Imperial Age, or upgrading a unit type. If a unit type is upgraded then all the current units of the specified type should change to the Elite version. The player should now be allowed to create Elite units of the type that was upgraded.
- `Talk(string)` - This is used for a player to talk on the GUI. Please keep all conversation civil. If you want to have some fun you can try discovering some of the easter eggs we added! (Come to Office Hours if you want to learn more)

The server should return `Success` or `Failed` depending on whether the action was legal or not. This is particularly necessary for teams that try to cheat! The client also has the ability to request information from the server about the state in order to update the game. These updates are requested with messages of type status. They must be handled in `handleStatus` in `game.ml` and must return the appropriate `Data` command for each status request.

- `UnitStatus unit_id`: Returns the `unit_data` for the unit identified by `unit_id`.
- `BuildingStatus building_id`: Returns the `building_data` for the building identified by `building_id`.
- `TeamStatus color`: Returns the `team_data` for the team labelled `color`.
- `ResourceStatus`: Returns a list of `resource_data` in order to describe all the available resources (their tile positions, the types of each resource, and their resource counts).
- `GameStatus`: returns all of the information related to the game as `game_data`.

6 GUI

6.1 GUI Client

In order to view the game, you will have to set up a GUI client program. The client has been coded for you, and is located in the gui directory. The client renders the game using Java and Swing. This module will be sufficient for simple rendering of the game, though you can make bonus modifications if you wish. To use the GUI, you must have Java.

The game server is responsible for sending graphical update messages to the GUI, as described below. The game server will listen for clients to send the updates to throughout the game. However, if a GUI connects halfway through the game, it will have missed the message that initializes the board. Therefore, if you do not start the process for the GUI before you start the processes for the teams, you will not be able to see anything from the GUI.

6.2 Sending messages to the GUI

We have provided a simple module called `Netgraphics` with functions to send graphical updates. The functions are specified in `netgraphics.ml`. Note that the `init` function is called by the `Server` module and `sendUpdates` is called regularly by the `Server` module. So in order to send a graphics update to the clients, the game module calls `Netgraphics.add_update` with the appropriate `update` type. The one exception to this is that the `InitGraphics`, `InitFood(tile_list)`, and `InitWood(tile_list)` updates must be sent by themselves, so the game module should send these updates directly by calling `Netgraphics.send_update`. The arguments `tile_list` for Food and Wood will always be `cFOOD_TILES` and `CWOOD_TILES`, respectively.

6.3 Graphics Commands

	Arguments	Meaning
`AddBuilding`	`building_id, building_type, tile, health, color`	Adds a building to the specified `tile`.
`AddUnit`	`unit_id, unit_type, position, health, color`	Adds a unit to the specified pixel `position`.
`DisplayString`	`color, string`	Adds the message to the GUI by the player labelled `color`
`DoAttack`	`unit_id`	Makes the unit identified by `unit_id` attack by playing its attack animation.
`DoCollect`	`unit_id, color, resource_type, amount_collected`	The GUI displays the wood and food counts for each team. This updates the GUI to increase the `resource_type` of the team labelled by `color` by `amount_collected`.
`GameOver`	`game_result`	Updates the GUI with the game result
`InitGraphics`		Tell the GUI client to initialize the graphics in preparation for a new game starting.
`InitFood`	`tile list`	Tell the GUI to initialize the food. Should be sent immediately after `InitGraphics`.
`InitWood`	`tile list`	Tell the GUI to initialize the wood. Should be sent immediately after `InitFood`.
`MoveUnit`	`unit_id, position list, color`	The GUI stores a list of all points for each unit through which the unit will move. At each time step the GUI moves each unit to the next point in their list. This update tells the GUI to add all the points in `position list` to the specified unit's path.
`RemoveUnit`	`unit_id`	Removes the specified unit from the GUI.
`RemoveBuilding`	`building_id`	Removes the specified building from the GUI.
`RemoveResource`	`tile`	Removes the resource located at `tile` from the GUI.
`StopUnit`	`unit_id, color`	Tells the GUI to stop moving the specified unit. All of the points in the unit's path are cleared.
`UpdateScore`	`color, score`	Updates the team color's score to `score`
`UpgradeAge`	`color`	Tells the GUI to upgrade the team color's age to the Imperial Age.
`UpdateBuilding`	`building_id, health`	Updates a building's health to `health`.
`UpdateResource`	`tile, resource_count`	Updates the resource located at `tile` to now only have `resource_count` food or wood left.
`UpdateUnit`	`unit_id, health`	Updates a unit's health to `health`.
`UpgradeUnit`	`unit_type, color`	Tells the GUI that, for the team specified by `color`, all units of `unit_type` are now upgraded to Elite.

Remember that the GUI knows very little about the game, so it is your responsibility to make sure that the GUI is updated properly!

7 Provided source code

Many files are provided for this assignment. Most of them you will not need to edit at all. In fact, you should only edit and/or create new files in the `game` and `team` directories (plus any edits you need to make to the compilation scripts). Here is a list of all the files included in the release and their contents.

`build*.bat`	Build scripts for the game server, teams, and GUI client (see Section 8)
`game/constants.ml`	Definitions of game constants
`game/definitions.ml`	Definitions of game datatypes
`game/game.mli`	Signature file for handling actions, time, rules, and the game state
`game/game.ml`	Stub file for actions, rules, and time
`shared/util.ml`	General use helper functions. You may want to familiarize yourself with the available functions, as there are many that may make debugging easier, or that may simplify other important functions you will write.

`game/netgraphics.mli`	Signature file for sending updates to the GUI client
`game/netgraphics.ml`	Implementation of sending updates to the GUI
`game/server.ml`	Starts the game server and deals with communication
`shared/connection.mli`	Signature file for connection helper module
`shared/connection.ml`	Implementation of connection helper module
`shared/thread_pool.mli`	Signature file for thread pool helper module
`shared/thread_pool.ml`	Implementation of thread pool helper module
`team/team.ml`	Basic framework for a client to interact with a game server
`team/babybot.ml`	Stub for a team AI

7.1 Code Structure

To understand how to implement the game, you must first understand how the code we have provided you operates. One crucial aspect is the relation between the `Server` module and the `Game` module. The `Server` module deals primarily with receiving connections from the teams and calls the `Game` module for all issues related to the game rules. *You do not need to modify the* `Server` *module, graphics commands, or GUI client.* Your modifications and additions will take place in the `Game` module, `State` module, and any other modules you choose to add.

7.2 Server

At a high level, `server.ml` does the following things:

1. Calls `Netgraphics.init`, which accepts connects from GUI clients
2. Waits until for enough teams to join the game (see Section 8.1 for more details)
3. Repeatedly calls `Game.handleTime` with the responses of the clients, outputting the state and request to the client
4. Uses the actions of team to call `Game.handleAction`
5. Uses the statuses of the team to call `Game.handleStatus`
6. Repeatedly sends graphics updates to the GUI

You will need to think carefully about how you design and implement the game to meet the `Server` module's expectations.

7.3 Game

All the functions in the `Game` module referred to above are specified in `game.mli`. Your design may require you to add or modify the type declarations or functions we have provided.

7.4 State

We strongly suggest that you have a `State` module in your final design, as the distinction between game rules and game state is significant. In other words, you should *not* put all of your code in the `Game` module.

8 Running the game

You will need to launch four command prompts to execute the game.

8.1 Game Server

From `ps6/game`, run `build_game.bat` or `build_game.sh` to build the game server. In this command prompt, run `game.exe`.

8.2 GUI

The GUI is already written for you in Java. It is distributed in a jar file called `gui_client.jar`. The GUI needs the `audio` and `images` folder for game assets. To run the GUI, just run the jar file (from the command line, `java -jar gui_client.jar`). Enter your Game Server connection information (by default it is localhost at port 10501), and press the `Connect` button.

8.3 Team

From `ps6/team` run `build_team.bat teamname` (or `build_team.sh teamname` for Mac/Linux systems) to build the bot in the file `bot/teamname.ml`. Next, run `teamname.exe localhost 10500`. Run this command in two separate command prompts, and watch the magic happen!

9 Your tasks

There are several parts to the implementation of this project. Make sure you spend time thinking about each part before starting. Start on this project *early*. There are many things you will have to take into consideration when designing the code for each section. Remember, **no extensions!**

9.1 Design meeting

Your first task is to create a design for your *Age of Upson* implementation and meet with the course staff to review it. Each group will use CMS to sign up for a meeting, which will take place between 18 April and 19 April. If you are unable to sign up for any of the available time slots on CMS, contact the course staff, and we will try to accommodate you.

At the meeting, you will be expected to explain the design of your system, explain what data structures you will use to implement the design, and hand in a printed copy of the signatures for each of the modules in your design. You are also expected to explain your initial thoughts about strategies for your player bot. In designing module interfaces, think about what functionality needs to go into each module, how the interfaces can be made as simple and narrow as possible, and what information needs to be kept track of by each module. Everyone in the group should be prepared to discuss the design and explain why the module signatures are the way they are. We will give you feedback on your design.

9.2 Implementing the game

Your second task is to implement the *Age of Upson* game in the file `game/game.ml` and any files you choose to add. Note that you should add files only to the `game` and `team` directories. You must implement the rules as described in Section 3 and handling of actions. You must also make sure that the actions teams make are rendered in the graphic display using the interface detailed in Section 6. You can use the sample team program we provide to test your game, but for full testing coverage you will need to write your own tests.

9.3 Designing a bot

Your third task is to implement a bot to play the game. A very weak bot that you can use as a basis for your bot code is provided. There are many different strategies for building a good bot. This is your chance to be creative and have fun creating a good AI. We will also provide some staff bots to test against, at our discretion. Further information on the server will be provided soon.

In the next couple weeks, we will release stronger bots for you to test your own against. Use this opportunity to evaluate your own bot, so you can create a bot that will do well at the tournament. Also, feel free to test your bots against other students. It can be used as an opportunity to test your game implementation and test your bot!

Remember that bots can send Talk messages to the game. This feature is mainly here for you to have fun with, since you can make your bots send audio messages as well (if you discover how). Please do not abuse this feature.

9.4 Documentation

Your final task is to submit a design overview document for this project. Since this project is both large and quite open-ended regarding the way one may choose to implement it, documentation becomes even more important. Your design overview document should cover *both* your implementation of the game itself and the bot you created. In discussing your bot, you should make note of what strategies you experimented with, and what you found to be most effective.

9.5 Things to keep in mind

Here are some issues to keep in mind when designing and implementing the game:

- **Do not modify any of the files in shared/ or gui/**. These files will be subject to change as we will be adding things onto the gui, and will be changing values of constants for the tournament.
- **You need to make a good design**. This project is both large and complicated; without spending time on making a design that is both solid and complete, you *will* very quickly get bogged down when you go to implement things. The importance of design cannot be overemphasized. Trying to write code before you have your design is a recipe for disaster on a project of this magnitude.

  Before writing any code, you should have a very clear idea of *all* of the following:

  - What concurrency issues exist and how to deal with them?
  - What information needs to be kept track of to fully represent the game?
  - How that information will be stored and accessed efficiently?
  - What the interface between your modules will be?
  - What invariants will hold between your modules?
  - Which modules will enforce those invariants?
- **Think carefully about how to break up your program into loosely coupled modules.** The program will be complex and difficult to debug unless you can develop modules that encapsulate important aspects of the game. Design the interfaces to these modules carefully so that you can work effectively with your partner and do unit testing of the modules as you implement.
- **Make sure that what is going on in the game matches what is going on in the graphics.** Updating one does not automatically update the other. If you are watching the game and something seems to go wrong, remember that it could just be the code controlling the output to the screen. Moreover, just because the graphics look correct doesn't mean the game is acting properly. It would behoove you to maintain some sort of invariant between the status of the game and the status of the graphics.
- **Problems in the game might actually be problems with the bots.** If you are using your own bots to test the actions and something seems wrong, the bots could just as easily be at fault.
- **Implement and test the actions one at a time.** Don't try to implement all of the actions and test them with one single bot. Start with easier actions and work up to the harder ones.
- **Remember to finish the game in time to write a good bot.** The bot part of the project is worth almost as much as the game part, so don't put off the bot part until the end. We WILL be grading it based on how well it performs against our test bots.

- If you have any ideas for a game for next semester for future students, then please e-mail jps327 or rak248.
- **The images and audio have been taken straight from the game Age of Empires II. They are not our property.** They belong to Microsoft.

9.6 Final submission

You will submit:

1. A zip file of all files in your `ps6` directory, including those you did not edit. We should be able to unzip this and run the `build_game.bat` (or `build_game.sh`) script to compile your game code, and the `build_team.bat` (or `build_team.sh`) script to compile your bot code (i.e., you should modify the scripts to include all necessary files). This should include:
    - Your game implementation
    - Your bot, named `bot.ml`, in the `team` directory along with any files it needs to build and run

   It is very important that you organize your files in this manner, as it greatly simplifies grading.
2. Your documentation file, in `.pdf` format.

Although you will submit the entire `ps6` directory, you should only add new files to the `game` and `team` folders; the other folders should remain unchanged. If you add new `.ml` or `.mli` files, you should add them to the compilation scripts. Note again that we expect to be able to unzip your submission and run the build scripts in the newly created directory to compile your code without errors or warnings. **Submissions that do not meet this criterion will lose points.**